MASD
pour HP 49G

Documentation originale par Cyrille de Brebisson

Traduit de l'anglais et adaptation par Philippe Pamart

1. Introduction

Faisant partie intégrante de la machine, une library cachée attends dans la Hp 49. Cette library contient beaucoup d'outils de développement principalement destinés à développer en Rpl-système et en assembleur. Pour utiliser cette library, vous devez l'attacher (256 ATTACH).
NB1 : vous pouvez aussi armer le flag –86. La library s'auto-attachera lors du prochain redémarrage à chaud de la machine.
NB2 : quand la library est attachée, elle apparaît dans le menu APPS.
NB3 : les outils et programmes de cette library sont extrêmement puissants, et une mauvaise utilisation peut conduite à une perte de la mémoire.

2. Les outils de la library de développement

2.1. APEEK

Commande PEEK sur une adresse : lit l'adresse stockée à une adresse.
Exemple : # 80711h APEEK retourne l'adresse de la racine HOME.

1: entier binaire ⇒ 1: entier binaire

2.2. PEEK

Lecture en mémoire : lit les quartets à une adresse spécifiée en mémoire.
NB : à cause du bank-switching, la lecture de données entre # 40000h et # 7FFFFh peut ne pas être exact.

2: entier binaire (adresse)
1: entier binaire (nombre de quartets à lire) ⇒ 1: chaîne de caractères

2.3. POKE

Ecriture en mémoire : écrit des quartets en mémoire.
NB1 : vous ne pouvez pas écrire en Flash-ROM en utilisant cette commande.
NB2 : écrire aléatoirement dans la mémoire provoquera des pertes de mémoire.

2: entier binaire (adresse où écrire)
1: chaîne de caractères (données à écrire) ⇒ 1: (rien)

2.4. A→

Retourne l'objet stocké à une adresse spécifiée.

1: entier binaire ⇒ 1: objet

2.5. →A

Retourne l'adresse d'un objet placé sur la pile.

1: objet ⇒ 1: entier binaire

2.6. →RAM

Similaire à NEWOB : cette commande fait une copie d'un objet en RAM, quelque soit l'objet. Cette commande permet de copier un objet placé en ROM en RAM.

1: objet ⇒ 1: objet placé en RAM

2.7. A→H

Adresse en chaîne de caractères : représente la forme hexadécimale d'une adresse (vous pouvez l'utiliser ensuite avec la commande POKE). La représentation hexadécimale est une chaîne de 5 caractères où l'adresse est écrite à l'envers.

1: entier binaire ⇒ 1: chaîne de caractères

2.8. H→A

Chaîne de caractères en adresse : retourne l'adresse représentée par une chaîne de 5 caractères. La représentation hexadécimale d'une adresse est une chaîne de 5 caractères où l'adresse est écrite à l'envers.

1: chaîne de caractères ⇒ 1: entier binaire

2.9. CD→

Code en hexadécimal : retourne la représentation hexadécimale d'un objet code (programme assembleur) sans prologue ni taille.

1: code ⇒ 1: chaîne de caractères

2.10. →CD

Hexadécimal en code : retourne l'objet code (programme assembleur) représenté par une chaîne de caractères.

1: chaîne de caractères ⇒ 1: code

2.11. →H

Objet en hexadécimal : retourne la représentation hexadécimale d'un objet.

1: objet ⇒ 1: chaîne de caractères

2.12. H→

Hexadécimal en objet : retourne l'objet représenté par une chaîne hexadécimale. Une chaîne hexadécimale est une chaîne de caractères qui contient uniquement les caractères '0' à '9' et 'A' à 'F'.
NB : si une chaîne de caractères ne représente pas d'objet valide, ça peut provoquer une perte de la mémoire.

1: chaîne de caractères ⇒ 1: objet

2.13. S→H

Chaîne en hexadécimal : retourne la représentation hexadécimale d'une chaîne de caractères.
Exemple : `"A" S¨H` donne `"14"`.

1: chaîne de caractères ⇒ 1: chaîne de caractères

2.14. H→S

Hexadécimal en chaîne : retourne la chaîne de caractères dont les données sont représentées par une chaîne hexadécimale. Une chaîne hexadécimale est une chaîne de caractères qui contient uniquement les caractères '0' à '9' et 'A' à 'F'.
Exemple : `"14" H¨S` donne `"A"`.

1: chaîne de caractères ⇒ 1: chaîne de caractères

2.15. SREV

Inversion de chaîne : inverse complètement une chaîne de caractères.
Exemple : "ABCDEF" SREV donne "FEDCBA".

1: chaîne de caractères ⇒ 1: chaîne de caractères

2.16. MAKESTR

Création d'une chaîne de caractères en fonction d'une taille donnée.
Exemple : 10 MAKESTR donne "ABCDEFG<cr>AB".

1: réel ⇒ 1: chaîne de caractères

2.17. SERIAL

Donne le numéro de série de votre Hp 49G.

1: (rien) ⇒ 1: chaîne de caractères

2.18. →S2

Décompile un objet en mode Rpl-système.
Exemple : « » ¨S2 donnera ceci : "!NO CODE !RPL :: x<< x>> ; @"

1: objet ⇒ 1: chaîne de caractères

2.19. XLIB~

Convertit des réels en objet XLIB.

2: réel ou entier binaire
1: réel ou entier binaire ⇒ 1: Xlib

2.20. CRC

Calcul de CRC : donne le CRC d'une chaîne de caractères ou d'une library. Cette commande vous donne le CRC des données d'une library ou d'une chaîne de caractères (le calcul du CRC commence à la taille de l'objet et finit quatre quartets avant la fin de l'objet).

1: chaîne de caractères ou library ⇒ 1: entier système

2.21. S~N

Chaîne en nom : cette commende permet de convertir une chaîne de caractères en nom global et vis versa. Cette commande permet de créer des noms non valides.
NB : ne pas effacer ou déplacer le répertoire vide placé à la racine (HOME). Ne pas modifier non plus aucune donnée dans ce répertoire.

1: chaîne de caractères ⇒ 1: nom global
ou bien
1: nom global ⇒ 1: chaîne de caractères

2.22. R~SB

Réel en entier système : cette commande permet de convertir un entier système en réel et vis versa.

1: réel ou entier ⇒ 1: entier système
ou bien
1: entier système ⇒ 1: réel

2.23. SB~B

Entier binaire en entier système : cette commande permet de convertir un entier système en entier binaire et vis versa.

1: entier binaire ⇒ 1: entier système
ou bien
1: entier système ⇒ 1: entier binaire

2.24. LR~R

Réel long en réel : cette commande permet de convertir un réel en réel long et vis versa.

1: réel ⇒ 1: réel long
ou bien
1: réel long ⇒ 1: réel

2.25. LC~C

Complexe long en complexe : cette commande permet de convertir un complexe en complexe long et vis versa.

1: complexe ⇒ 1: complexe long
ou bien
1: complexe long ⇒ 1: complexe

2.26. COMP→

Décomposition : en Rpl, c'est l'équivalent à la commande `LIST¨`, mais ne fonctionne que sur les programmes et les objets symboliques.

n+1 à 2 : objets
1: liste/programme/objet symbolique ⇒ 1: réel (n = nombre d'objets)

2.27. →ALG

Création symbolique : en Rpl, c'est l'équivalent à la commande `¨LIST`, mais ne crée que des objets symboliques.

n+1 à 2 : objets symboliques
1: nombre d'objets (réel) ⇒ 1: expression symbolique

2.28. →PRG

Création de programme : c'est l'équivalent à la commande `¨ALG`, mais ne crée que des programmes.

n+1 à 2 : objets
1: nombre d'objets (réel) ⇒ 1: programme

2.29. →LST

Création symbolique : en Rpl, c'est l'équivalent à la commande `¨LIST`, mais convertit un programme ou un objet symbolique en liste.

n+1 à 2 : objets
1: nombre d'objets (réel) ⇒ 1: liste
ou bien
1: programme/liste/objet symbolique ⇒ 1: liste

3. CRLIB

3.1. Construction d'une library

Commande de création de library.
Une library est un des objets les plus complexes dans la Hp 49G. Un des usages courants d'une library est de regrouper tous les fichiers d'une application.

Pour créer une library, vous devez stocker toutes les variables qui feront partie de la library dans un répertoire. Et ensuite, vous devez configurer votre library en créant des variables spéciales.

La variable `$TITLE` doit contenir une chaîne de caractères qui sera le titre de la library. Cette chaîne de caractères doit impérativement faire moins de 256 caractères. Les cinq premiers caractères définiront le nom de la library qui sera affiché dans le menu library.

La variable `$ROMID` contiendra le numéro de votre library. Ce nombre doit être compris entre 769 et 1791. Afin d'éviter des conflits avec d'autres library, vous devriez aller faire un tour sur **www.hpcalc.org** pour voir si le numéro de library que vous avez choisi n'existe pas déjà.

La variable `$CONFIG` contient l'objet de configuration de la library qui sera exécuté après un redémarrage à chaud. L'action la plus simple que ce programme peut faire est d'attacher la library au répertoire HOME. En plaçant un entier ou un réel dans la variable `$CONFIG` provoquera une erreur lors de l'exécution de `CRLIB`. Ce programme doit laisser la pile intacte et ne doit pas provoquer d'erreurs.

La variable `$VISIBLE` doit contenir une liste de toutes les variables du répertoire qui doivent apparaître visibles dans le menu de la library.

La variable `$HIDDEN` doit contenir la liste de toutes les variables du répertoire qui doivent apparaître invisibles dans le menu de la library. Ce sont en général les sous-programmes de votre application.

La variable `$EXTPRG` doit contenir le nom de l'extension de programme de la library. Ce programme peut être soit un objet visible ou invisible de la library. Voir le paragraphe qui lui est consacré juste après.

Enfin, quand vous avez défini toutes ces variables, vous pouvez exécuter `CRLIB` pour créer la library qui sera déposée sur le niveau 1 de la pile.

3.2. Extension de programme

Il est possible d'agrandir le menu statistiques en utilisant la library utilisateur. La Hp 49 permet de personnaliser un menu avec des fonctions, comme si elle faisaient partie de la machine.

Exemple : personnalisation du menu principal statistiques.
Basculer en mode Rpl (MODE ▶ +/- ENTER) et attacher la library de développement (`256 ATTACH`).
Dans un répertoire, créer les variables suivantes :

```
$ROMID     1324
$CONFIG    1
$TITLE     "Personnalisation"
$VISIBLE   { ABOUT }
$HIDDEN    { MessageHandler }
$EXTPRG    'MessageHandler'
ABOUT      "Cette library est un exemple de personnalisation"
MessageHandler
« IF DUP 1 R~SB ==
  THEN SWAP
  { { "7 Nouvelle entrée" « "Mes stats" 1 DISP 7 FREEZE » } } +
  SWAP
  END »
```

Créer la library (`CRLIB`) et la stocker dans un port d'extension (ex. : `0 STO`).

Maintenant, il reste juste à entrer dans le menu "statistiques" (▶ 5) !

Comment ça marche ?
Chaque fois que le menu STAT est appelé, la Hp 49 lance chaque programme extension de la library dans le système. Ce programme extension prend un numéro de message sur la pile (et le laisse sur la pile !). Chaque numéro de message a une spécification, comme ci-dessous :

Voici les entrées/sorties attendues par le programme d'extension pour les différents menus :

Menu APPS :
Entrée { { "String" Action } ... } ZERO
Sortie Liste modifiée ZERO

Menu APPS :
Entrée { { "String" Action } ... } ZERO
Sortie Liste modifiée ZERO

Menu STAT :
Entrée { { "String" Action } ... } ONE
Sortie Liste modifiée ONE

Menu Hypothesis Statistic :
Entrée { { "String" Action } ... } TWO
Sortie Liste modifiée TWO

Menu Confidence Interval Statistic :
Entrée { { "String" Action } ... } THREE
Sortie Liste modifiée THREE

Menu Finance :
Entrée { { "String" Action } ... } FOUR
Sortie Liste modifiée FOUR

Menu Numeric Solver :
Entrée { { "String" Action } ... } FIVE
Sortie Liste modifiée FIVE

4. Le compilateur de langage machine MASD

4.1. Généralités sur le langage machine

Comme le processeur Saturn exécute directement des instructions en langage machine, le système d'exploitation ne peut pas contrôler ce qu'un programme en langage machine fait. Sur la Hp 49, les données utilisateurs sont stockées dans la même zone qu'un objet temporaire. Quand il y a un bug dans un programme en langage machine, vous avez de bonnes chances de perdre vos données. Soyez donc très prudent quand vous programmez en langage machine.
Le langage machine est un langage dépendant du processeur, donc ce que vous apprendrez sur la Hp 49 ne pourra pas être utilisé sur un autre processeur. D'un autre côté, les techniques de programmation ne dépendent pas du hardware et peuvent donc être réutilisées.

4.2. Lancement de MASD

Pour compiler un programme, poser une chaîne de caractères sur le niveau 1 de la pile et taper `ASM` ou utiliser le menu `ASM` de la library DevTools (library 257).

4.3. Généralités sur la syntaxe MASD

MASD attend une chaîne de caractères (appelée source) sur le niveau 1 de la pile.
Une source est une ensemble d'instructions, de commentaires, d'espaces et de fins matérialisées par des retours chariot, et du signe @.
MASD est un outil sensible, soyez donc prudent, comme par exemple avec "`boucle`" et "`BOUCLE`" qui sont deux labels différents.

Les caractères de séparation sont le caractère espace (code ASCII=32). Ils incluent les espaces, les tabulations, les sauts de ligne et les retours chariot.
Certaines instructions on besoin d'un paramètre appelé champ. Les caractères de séparations entre une instruction et un champ peuvent être l'espace, la tabulation et le point. Exemple : `A+B.A` peut être utilisé à la place de `A+B A`.
Les commentaires peuvent être insérés n'importe où entre deux instructions. Ils commencent avec un % ou un `;` et finissent en fin de ligne courante.
Des directives change la manière de MASD à interpréter votre source. Ces instructions commencent avec un `!`, mais nous l'expliquerons plus tard.

4.4. Les erreurs

Si MASD détecte un ou plusieurs erreurs de syntaxe, il posera la pile une liste détaillant toutes les erreurs. Cette liste est utilisée par ER pour trouver l'emplacement des erreurs, et laisse l'utilisateur les corriger.
Les erreurs peuvent être annoncées à la mauvaise place. Par exemple, si un label n'est pas correctement défini, les erreurs seront situées sur les instructions appelées.
La liste d'erreurs suit un format strict :
C'est une liste avec au plus 16 sous listes.
Chaque sous liste contient 2 entiers système et un nom global.
Le premier entier système le numéro du message d'erreur.
Le second un entier système utilisé pour indiquer de combien le saut est trop grand.
Le troisième indique la position de l'erreur dans la source.
Le nom global un NULLNAME si l'erreur est dans la source principale, ou le nom de fichier de la source buguée.

Le programme ER prend 2 objets comme arguments : la source originale du code sur le niveau 2 et la liste d'erreurs sur le niveau 1. Normalement, vous devriez compiler votre source en utilisant un procédé semblable à : `IFERR ASM THEN ER END`.

4.5. Les liens

Les liens sont des fichiers sources qui peuvent être attaché à la source principale pendant la compilation (équivalent à {$I} en Pascal ou à #include en C).
Quand un lien est appelé, MASD arrête de compiler le fichier principal et se lance dans la compilation du fichier lié, et une fois ce lien compilé il revient au fichier principal pour continuer la compilation.
La syntaxe en mode ASM :
`'FileName` appelle le fichier nommé FileName
La syntaxe en mode RPL :
`INCLUDE FileName` appelle le fichier nommé FileName
NB1 : un lien peut appeler d'autres liens.
NB2 : vous ne pouvez pas utiliser plus de 64 liens dans votre projet.
NB3 : pour savoir comment MASD recherche les fichiers, voir la section qui s'en occupe.
NB4 : les liens sont souvent utilisés pour décomposer un projet en parties indépendantes afin de permettre un accès plus rapide et plus facile au code source.

4.6. Utilisation des labels

Un label est une étiquette dans le programme. La principale utilisation des labels est de déterminer la destination des sauts. Un label est défini sur moins de 128 caractères différents de l'espace, `+`, `-`, `*` et `/`. Un label commence avec une étoile * et finit avec un caractère de séparation (un espace pour l'essentiel).
Syntaxe en mode ASM :
`*BigLoop` correspond à la déclaration du label
Syntaxe en mode RPL :
`LABEL BigLoop` correspond à la déclaration du label
Soyez prudent sur l'utilisation des labels.
Trois types de label peuvent être utilisés :

- les labels globaux : un label global peut être utilisé n'importe où dans le projet, comme les variables globales en Pascal ou en C. Exemple : `*TOTO`
- les labels locaux : un label local est seulement utilisé dans une source locale, comme les variables locales en Pascal ou en C. Une section locale commence au début de la source locale, après un label

global et après un lien (voir section parlant des liens). Une section locale finit en fin de source locale, avant un lien et avant un label global. Un label local est défini par le caractère `.` comme premier caractère. Exemple `*.TOTO`
- les labels de liens : un label de lien est un label qui existe seulement dans le lien où il a été déclaré. Un label de lien est défini par le caractère _ comme premier caractère. Exemple : `*_TOTO`

NB : dans les projets, utiliser moins de labels globaux est mieux car un label global est plus long à compiler et parce qu'il donne une meilleure structure de programme. La bonne habitude est d'utiliser les labels globaux pour couper un programme en sous-routines et d'utiliser les labels locaux dans ces sous-routines.

4.7. Utilisation des constantes

Il est possible de définir des constantes. C'est utilisé pour définir des adresses mémoire par un nom, plutôt que par son adresse elle-même.
Par exemple, au lieu de taper `D1=80100` à chaque fois que c'est nécessaire, c'est mieux de déclarer `DC Result 80100` au début du projet et ensuite de taper `D1=(5)Result` à chaque appel de cette adresse.

Déclaration des constantes :
`DC` *`CstName ExpressionHex`* ou
`EQU` *`CstName ExpressionHex`* ou
`DEFINE` *`CstName ExpressionHex`*.
`ExpressionHex` est un nombre hexadécimal ou une expression (commençant par un caractère et qui ne peut commencer par un nombre hexadécimal). Une expression qui commence par un nombre hexadécimal peut être saisie si elle commence par le caractère $, une expression commençant avec un nombre décimal peut être saisie si elle commence par le caractère #.
NB1 : une constante ne peut pas avoir le même nom qu'un label qui est déjà déclaré.
NB2 : le nom de constante suit le mêmes règles que le nom de label.
NB3 : la valeur d'une constante est stockée sur 16 quartets.

MASD introduit le pointeur de constantes appelé CP qui aide à définir les constantes. CP est défini par : `CP=ExpressionHex`
CP est défini sur 5 quartets et sa valeur initiale est 80100.

`DCCP` *`Increment ConstantName`*
déclare une constante en fonction de la valeur courante de CP et incrémente CP avec l'incrément.
NB : l'incrément est une valeur hexadécimale, pour utiliser une valeur décimale, utiliser un # comme premier caractère.

Comme exemple, si CP est égal à $10
`DCCP 5 Foo`
definit la constante `Foo` avec une valeur de $10 et incrémente la valeur de CP à $15

Plusieurs constantes peuvent être définies, en commençant à partir de CP.
`:` *`Inc CstName0 CstName1 … CstNameN-1`* `:`
définit *N* constantes *`CstNameX`* avec une valeur de *CP+X*Inc* et change la valeur de CP à *CP+N*Inc*.
Par défaut, *`Inc`* est un nombre décimal. Il peut être saisi comme un nombre hexadécimal avec $ comme premier caractère.

4.8. Les expressions

Une expression est une opération mathématique qui est calculée à la compilation.
Les termes de cette opérations sont des valeurs hexadécimales ou décimales, des constantes ou des labels.
Une expression prend fin sur un caractère de séparation.
```
DCCP 5 @Data
...
D1=(5)@Data+$10/#2  D0=(5)$5+DUP  LC(5)"DUP"+#5
```
sont des expressions correctes.

NB1 : une valeur hexadécimale doit commencer par le caractère $.
NB2 : une valeur décimale doit commencer par le caractère # ou directement le nombre.

NB3 : un & ou(*) sont équivalents à l'adresse de l'instruction courante en mode absolu, ou à l'offset courant en mode standard (cette valeur ne signifie rien en elle-même, mais peut-être utilisée pour calculer la distance entre un label et une instruction courante).
NB4 : la valeur d'un label est l'adresse du label en mode absolu, ou l'offset si le label dans le programme est en mode normal.
NB5 : les entrées de EXTABLE peuvent être utilisées. Dans un cas spécial (`DUP+#5` peut être une addition, ou une entrée `'DUP+#5'`), ajouter `""` autour du mot donne : `"DUP"+#5`.
NB6 : les calculs sont réalisés sur 64 bits (16 quartets).
NB7 : X divisé par 0 donne `$FFFFFFFFFFFFFFFF`.
NB8 : pour éviter d'utiliser de la mémoire pour rien, MASD essaie de compiler les expressions dès qu'il les voit. Si MASD n'est pas capable de compiler une expression directement, elle est compilée en fin de compilation. Toujours pour utiliser moins de mémoire, il est préconisé de déclarer vos constantes en début des sources.
NB9 : MASD peut forcer à compiler des expressions en utilisant la directive `!COMPEXP`.
NB10 : les seuls symboles non permis dans les labels sont +, -, * et /, par ailleurs si vous désirez utiliser un symbole opérateur après un label, vous devez mettre le symbole entre " pour limiter ce symbole. Exemple : `"DUP"<<5`.
NB11 : un label avec d'étranges caractères peut être placé entre ".
NB12 : si vous choisissez d'utiliser des nombres signés dans les expressions, soyez très prudent !
NB13 : La pile d'évaluation des expressions de MASD vous permet d'avoir approximativement 10 calculs en attente. Exemple: 1^2&3-5*9%7 s'évalue 1^(2&(3-(5*(9%7)))) et utilise 4 niveaux de pile
NB14 : si vous utilisez des labels de liens ou locaux dans une expression qui ne peut être assemblée tout de suite, vous devez ajouter un `!COMPEXP` avant la fin des limites du lien. Dans une expression, vous ne pouvez pas utiliser un label local indéfini et un label local indéfini en même temps.

MASD reconnaît ces opérateurs :

Opérateur	Priorité	Notes
<<	7	Left Shift : 1<<5 = $20
>>	7	Right shift : $20>>5 = 1
%	6	Modulo (reste de la division) X%0=0
*	5	Multiplication
/	5	Division X/0=$FFFFFFFFFFFFFFFF
+	4	Addition
-	4	Soustraction
<	3	Inférieur (true = 1, false = 0)
>	3	Supérieur (true = 1, false = 0)
<=, ≤	3	Inférieur ou égal (true = 1, false = 0)
>=, ≥	3	Supérieur ou égal (true = 1, false = 0)
=	3	Egalité (true = 1, false = 0)
#, ≠	3	Différent (true = 1, false = 0)
&	2	Et logique
!	1	Ou logique
^	1	Ou exclusif

4.9. Les sauts

Les sauts sont un premier pas du langage machine vers un langage de 3ème génération, même s'il y a un autre moyen d'écrire des instructions SATURN.
La base du saut est la structure bloc.
Un bloc est fermé par deux accolades { et }, et peut être insérée dans un autre bloc.

Les instructions suivantes traitent avec les blocs :

SKIPS instructions	**Equivalents**
`{ … }`	définit un bloc (ne génère pas de code)
`SKIP { … }`	`GOTO .S ... *.S`
`SKIPL { … }`	`GOTOL .S ... *.S`
`SKIPC { … }`	`GOC .S ... *.S`
`SKC { … }`	`GOC .S ... *.S`
`SKIPNC { … }`	`GONC .S ... *.S`

SKNC { … }	GONC .S ... *.S
Test SKIPYES { … }	Test GOYES .S ... *.S
Test { … }	Test GOYES .S ... *.S
Test ¨ { … }	/Test GOYES .S ... *.S (/Test est un test opposé)
Test -> { … }	/Test GOYES .S ... *.S
SKUB { … }	GOSUB .S ... *.S
SKUBL { … }	GOSUBL .S ... *.S
STRING { … }	$/02A2C GOIN5 *.S ... *.S (pour créer une chaîne de caractères)
CODE { … }	$/02DCC GOIN5 *.S ... *.S (pour créer un objet code)
STROBJ $*PROLOG* { … }	$(5)*PROLOG* GOIN5 .S ... *.S (pour créer un 'prologue-longueur' d'objet)

/Test est l'opposé d'un test. Par exemple si un Test est ?A<C.A, /Test devient ?A>=C.A. Les instructions de test traitant aussi avec le registre hardware (?HST=0, ?MP=0, ?SR=0, ?XM=0 et ?SB=0) ne peuvent être inversées.

Une fois les blocs définis, des instructions spéciales peuvent être utilisées dessus. Ces instructions appelées EXIT et UP permettent de sauter à la fin ou au début d'un bloc.

Ces instructions	sont équivalentes à
{ EXIT EXITC EXITNC ?A=0.A EXIT UP UPC UPNC ?A=0.A UP }	*.Beginning GOTO.End GOC.End GONC.End ?A=0.A ¨.End GOTO.Beginning GOC.Beginning GONC.Beginning ?A=0.A ¨.Beginning *.End

NB : ne pas confondre entre les instructions EXIT et UP qui sont des GOTOs, et EXIT et UP après un test qui sont des GOYESs.

EXIT et UP peuvent sauter au début ou à la fin d'un niveau de bloc supérieur en spécifiant le numéro du bloc à quitter, après les instructions UP ou EXIT.

Ces instructions	sont équivalentes à
{ { { UP2 UP3 EXIT1 EXIT3 } } }	*.Beg3 *.Beg2 *.Beg1 GOTO.Beg2 GOTO.Beg3 GOTO.End1 GOTO.End3 *.End1 *.End2 *.End3

NB : EXIT1 est équivalent à EXIT, et UP1 à UP.

En utilisant les instructions SKELSE, SKEC, SKENC, SKLSE, deux blocs créent une structure du type IFNOT-THEN-ELSE.

Ces instructions	sont équivalentes à	ou dans un langage de haut niveau
?A=0.A SKIPYES { EXIT UP } SKELSE { A+1.A EXIT UP }	?A=0.A GOYES.Beg2 *.Beg1 GOTO.End2 % et pas End1 GOTO.Beg1 *.End1 GOTO.End2 *.Beg2 A+1.A GOTO.End2 GOTO.Beg2 *.End2	IF NOT A=0 THEN BEGIN ... ... END ELSE BEGIN ... ... END

NB1 : SKELSE place un GOTO entre deux blocs, SKEC place un GOC, SKENC place un GONC et SKLSE ne place rien.
NB2 : les UPs sont compilés tout de suite tant que les EXITs et les blocs ouverts sont gardés dans une pile. Vous ne pouvez pas avoir plus de 64 objets dans la pile.

4.10. Macros et insertions

Si des données doivent être insérées dans un projet, elles peuvent être entrées en hexadécimal dans un fichier source, en utilisant $.
Mais le moyen le plus simple est d'inclure des données par le biais d'un fichier externe à la source, qu'on appelle une macro. Un fichier macro doit être une chaîne de caractères, un graphique, un objet code, ou une liste.
Dans le cas d'une chaîne ou d'un code, MASD insèrera seulement la partie données (après la longueur). Dans le cas d'un graphique, seules les données graphiques seront insérées (pas de longueur, pas de dimensions).
Dans le cas d'une liste, seul le premier terme de la liste sera inclus en suivant les règles précédentes.
La syntaxe est :
`/FileName`
NB : pour savoir comment MASD recherche le fichier FileName, regarder les paragraphes suivants.

Vous pouvez inclure un objet complet en utilisant `INCLUDE` ou `INCLOB`.
En mode ASM, utilisez `INCLUBE` ou `INCLOB` suivi d'un nom de fichier pour inclure un objet. En mode RPL utilisez `INCLOB`.

4.11. Convention sur les noms de fichiers

MASD a parfois besoin de rechercher un fichier dans la mémoire de la Hp 49. Le fichier peut être trouvé soit en spécifiant l'emplacement et le nom du fichier, ou seulement le nom de fichier qui doit être cherché dans la liste de recherche des chemins.
La première liste de chemins contient le répertoire courant, le répertoire parent et le répertoire HOME.
NB : vous pouvez ajouter un répertoire dans la liste de recherche des chemins en utilisant `!PATH+` *RepName* où *RepName* est un nom de répertoire les règles du nom de chemin complet, chose expliquée plus bas.
Pour définir un chemin complet, utiliser :
`H/` pour définir HOMEDIR comme la racine.
x/ où x est un numéro de port , afin de définir un port comme racine.
La racine est suivie par une liste de répertoires finissant avec le nom du fichier.
`2/TOTO/TITI/TUTU` définit le fichier TUTU dans le répertoire TITI, puis dans TOTO du port n°2.
`H/ME/YOU` définit le fichier YOU stocké dans le répertoire ME, dans HOMEDIR.
NB1 : vous pouvez la notation d'un répertoire complet comme un paramètre pour `!PATH+`.
NB2 : vous ne pouvez pas définir plus de 16 entrées dans le recherche d'un répertoire.

4.12. Les Tests

Une instruction test (`?A=0.A`) peut être suivie par un couple de choses différentes :
- un `GOYES Label`, `¨ Label` ou `-> Label`.
- un `-> {` ou `¨ {`. Dans ce cas, le test est inversé et un saut de bloc est ouvert.
- un `RTY` ou `RTNYES`.
- un `SKIPYES {` ou `{`. Dans ce cas, un saut de bloc est ouvert.
- un `GOTO`, `GOTOL`, `GOVLNG`, `GOSUB`, `GOSUBL` ou `GOSBVL`. Dans ce cas, le test est inversé et une instruction propre au saut est générée.
- un `EXIT` ou `UP`.

4.13. Le mode SysRpl

MASD peut basculer en mode SysRpl (aussi appelé Rpl-système ou External) en utilisant l'instruction `!RPL`. Dans le mode Rpl, beaucoup de changements ont eu lieu.
NB : Si le flag –92 est armé (`-92 SF`), MASD démarre avec le mode `!RPL` et `!NO CODE`.

La séparation des caractères est toujours effectuée grâce à l'espace.

Les commentaires commencent avec * en début de ligne et finissent en fin de ligne, ou commencent avec (et finissent avec).

4.13.1. Les instructions

En mode Rpl, MASD interprète les instructions dans l'ordre suivant :

4.13.1.1. Les constantes

Si une constante ou un label déjà défini existe avec le même nom, la valeur de la constante est utilisée sur 5 quartets.

Exemple :
```
EQU TOTO 2F034
...
TOTO
```
qui donnera :
```
430F2
```

4.13.1.2. External (dans la table des points d'entrée)

Si une entrée de la table des externals existe avec le même nom, la valeur associée à l'entrée sera utilisée.

Exemple :
```
TURNMENUOFF
```
donnera :
```
430F2
```
La table des points d'entrée est la library 258.

NB1 : Utilisez la TableCreator et le logiciel SDK pour la Hp 49 sur PC pour créer la table.
NB2 : Utiliser la table des externals est plus rapide que d'utiliser les constantes. D'un autre coté, les constantes sont un projet à charge, ce qui n'est pas le cas de la table des externals.

4.13.1.3. Les instructions MASD

`::`	Prologue d'un programme `$02D9D`
`;`	Epilogue d'une liste, d'un programme ou d'une expression algébrique `$0312B`
`{`	Prologue d'une liste `$02A74`
`}`	Epilogue d'une liste `$0312B`
`SYMBOLIC`	Prologue d'une expression algébrique `$02AB8`
`UNIT`	Prologue d'un objet unité `$02ADA`
`FPTR x y` ou `FPTR2 ^constant`	Un pointeur de flash est inséré (quelques opérations sont pratiquées sur la constante pour la 'développer' dans un pointeur de flash).
`# cst`	Entier système *cst* donné en hexadécimal.
`PTR cst`	Adresse. `PTR 4E2CF` donne `FC2E4`.
`ACPTR cst1 cst2`	Pointeur étendu.
`ROMPTR LibN ObjN`	XLIB.
`% real`	Nombre réel.
`%% real`	Nombre réel long.
`C% real1 real2`	Nombre complexe.
`C%% real1 real2`	Nombre complexe long.
`"..."`	Chaîne de caractères. Les caractères spéciaux peuvent être insérés en tapant \ et la valeur du code ASCII sur deux caractères hexadécimaux.
`ID name`	Nom global.
`LAM name`	Nom local.
`TAG chrs`	Objet tag.

X*xlibName*	XLIB identifié par son nom. Si c'est une commande standard Hp 49 (comme xDUP), l'adresse est utilisée à la place d'un objet XLIB.
HXS *Size Data*	Entier binaire ($02A4E), avec la taille *Size* en hexadécimal et les données *Data* sous forme d'une suite de caractères hexadécimaux. Exemple : HXS 5 FFA21
GROB *Size Data*	GROB ($02B1E).
LIBDAT *Size Data*	Library data ($02B88).
BAK *Size Data*	Backup ($02B62).
LIB *Size Data*	Library ($02B40).
EXT1 *Size Data*	Extended1 ($02BAA).
EXT2 *Size Data*	Extended2 ($02BCC).
EXT3 *Size Data*	Extended3 ($02BEE).
EXT4 *Size Data*	Extended4 ($02C01).
ARRAY *Size Data*	Array ($029E8).
LNKARRAY *Size Data*	Linked Array ($02A0A).
CODE *Size Data*	Code object ($02DCC).
CODE Assembly stuff ENDCODE	Insère un objet code, bascule en mode assembleur et ferme l'objet code sur le prochain ENDCODE.
NIBB *Size Data* or NIBHEX *Data* or CON(*Size*) *Expr*	Insère directement des données hexadécimales (pas de prologue).
CHR x	Objet caractère.
INCLOB *FileName*	Insère le contenu du fichier *FileName*.
INCLUDE *FileName*	Insère la source du fichier *FileName* pour être compilée (comme ' en mode assembleur).
LABEL *label*	Déclare un label (comme * en mode assembleur).
EQU *CstName ExpHex* ou DEFINE *CstNam ExpHex*	Déclare une constante (comme DC en mode assembleur).
EQUCP *Interleave CstName*	Déclare une constante (comme DCCP en mode assembleur).

NB : une constante peut être déclarée en mode ASM ou RPL, et peut être utilisée dans les deux modes.

4.13.1.4. Valeur décimale (entier système)

Si une instruction n'est pas encore reconnue, et si c'est une valeur décimale, MASD génère un entier système.

4.13.1.5. Variables locales sans nom

MASD essaie de varier les instructions en déclarant des variables locales.
Un environnement local est utilisé comme ceci :
{{ var1 var2 … var*N* }} avec *N*<23.
Ces variables ont un nom pendant la compilation, mais sont considérées comme variables locales sans nom, variables qui sont plus rapide que des variables locales nommées.
Une variable locale est rappelée en tapant son nom.
Des données peuvent être stockées dans une variable locale en tapant son nom, avec en entête ! ou =.
NB1 : une variable locale est valable jusqu'à la prochaine définition variables locales.
NB2 : l'environnement des variables locales n'est pas fermé automatiquement, utiliser ABND ou d'autres mots fournis.

Exemple :
{{ label1 label2 .. labelN }} deviendra :
' NULLLAM <#N> NDUPN DOBIND (or 1LAMBIND si c'est seulement une variable)
et :
label1 → 1GETLAM

```
=label1 → 1PUTLAM
!label1 → 1PUTLAM
```

Exemple de programme :

`::` `{{ A B }}` `B A!` `ABND` `;`	`::` `' NULLLAM TWO NDUPN DOBIND` `2GETLAM 1PUTLAM` `ABND` `;`

NB : en mode RPL, MASD bascule en mode ASM en utilisant la directive `!ASM`.

4.13.2. Exemple de programme

Comme en mode ASM, une source RPL doit se finir par un @.

```
"!RPL !NO CODE
::
ONE
CODE
SAVE LOADRPL
ENDCODE
TWO
xDUP2
;
@"
```

4.14. Les mnémoniques du SATURN

Dans cette section :
- x est un entier compris entre 1 et 16,
- h est un chiffre hexadécimal,
- a est un nombre compris entre 1 et 16 ou entre 0 et 15 selon le mode courant (0-15 ou 1-16),
- f est un champ (A, B, X, XS, P, WP, M ou S),
- v prend la valeur 0 ou 1,
- Reg est un registre de calcul (A, B, C ou D),
- SReg est un registre de sauvegarde (R0, R1, R2, R3 ou R4),
- Exp est une expression,
- Cst est une constante. La valeur est donnée en hexadécimal ou en décimal en utilisant respectivement l'entête $ ou #,
- Dreg est un pointeur (D0 ou D1),
- Data est une donnée pointée par D0 ou D1, soit DAT0 ou DAT1.

NB1 : Pour les instructions qui utilisent deux registres de calcul, seules les couples suivants sont valables : (A et B), (B et C), (C et D) et (A et C).
NB2 : Pour les instructions du type Reg1=Reg1... on peut écrire seulement Reg1... Exemple : A=A+C.A est équivalent à A+C.A.

4.14.1. Mise à zéro d'un registre

Syntaxe : Reg=0.f
Exemple : A=0.M

4.14.2. Charger une valeur dans le registre A ou C

LA et LC permettent de charger une constante dans le registre A ou C.
Syntaxe : LC hhh...hh charge x quartets dans C,
LA hhh...hh charge x quartets dans A.
Exemple : LC 80100

NB : LC #12 permet de charger l'entier décimal 12 sur les 3 premiers quartets de C. Le nombre de quartets utilisé est le nombre de chiffres que comporte la valeur (incluant le #). C'est à dire que LC #12 prendra 3 quartets. Donc, LC #12 correspond à LC 00C.

Syntaxe : LCASC(x) Characters charge dans C la valeur hexadécimal de x caractères. x doit être compris entre 1 et 8. LAASC(x) fait la même chose pour le registe A.
Exemple : LCASC(7) HP_MASD

Syntaxe : LC(x) Exp ou LA(x) Exp charge une expression dans C ou A en utilisant x quartets.
Exemple : LC(5)@Buffer+DataOffset

4.14.3. Changement de l'état d'un bit pour A et C

Syntaxe : RebBIT=v.a
Exemple : ABIT=1.5

4.14.4. Charger la valeur d'un registre dans un autre registre

Syntaxe : Reg1=Reg2.f
Exemple : A=B.X

4.14.5. Echange entre deux registres

Syntaxe : Reg1Reg2EX.f
Exemple : CDEX.W

4.14.6. Addition

Syntaxe : Reg1=Reg1+Reg2.f ou Reg1+Reg2.f
Exemple : C=C+A.A ou C+A.A
NB : si Reg1 et Reg2 sont le même registre, alors on multiplie ce registre par 2, comme par exemple A+A.A

4.14.7. Soustraction

Syntaxe : Reg1=Reg1-Reg2.f ou Reg1-Reg2.f
Exemples : C=C-B.A ou C-B.A
NB : les instructions suivantes fonctionnent aussi : A=B-A.f B=C-B.f C=A-C.f D=C-D.f

4.14.8. Incrémentation (+x) et décrémentation (-x)

Syntaxe : Reg=Reg+Cst.f ou Reg+Cst.f
Reg=Reg-Cst.f ou Reg-Cst.f
Exemples : A=A+50.A ou A+50.A
A=A-10.X ou A-10.X

NB1 : le Saturn n'est pas capable d'ajouter une constante plus grande que 16 à un registre, mais si Cst est plus grand alors MASD génèrera automatiquement les instructions nécessaires (exemple : A+20.A donnera A+16.A A+4.A).
NB2 : même si on additionne souvent des constantes à des registres, les grosses constantes doivent être évitées car elles ralentiront le programme et génèrerons aussi un gros programme.
NB3 : ajouter une constante plus grande que 1 à un registre sur le champ P, WP, XS ou S ne fonctionne pas correctement : c'est un bug livré avec le Saturn (problème de propagation de la retenue CARRY). Utiliser ces instructions avec prudence !
NB4 : après avoir ajouté une constante plus grande que 16 à un registre, la retenue CARRY ne pourra être testée.
NB5 : vous pouvez utiliser une expression plutôt qu'une constante (MASD doit être capable d'évaluer l'expression). Si l'expression est négative, MASD inversera l'addition en soustraction et vice versa.

NB6 : soyez prudent quand vous utilisez la soustraction, c'est facile de se tromper ! `A-5-6.A` est équivalent à `A+1.A` et non à `A-11.A` !!!
NB7 : S'il y a un dépassement, la retenue CARRY passe à 1.

4.14.9. Décalage d'un quartet à droite (division par 16)

Syntaxe : `RegSR.f`
Exemple : `ASR.W`
NB : si des bits sont perdus, SB passe à 1. Attention à réinitialiser SB pour les calculs suivants, car il ne se réinitialise pas automatiquement.

4.14.10. Décalage d'un quartet à gauche (multiplication par 16)

Syntaxe : `RegSL.f`
Exemple : `ASL.W`
NB : S'il y a un dépassement, la retenue CARRY passe à 1.

4.14.11. Décalage d'un bit à droite (division par 2)

Syntaxe : `RegSRB.f`
Exemple : `ASRB.W`
NB : si des bits sont perdus, SB passe à 1. Attention à réinitialiser SB pour les calculs suivants, car il ne se réinitialise pas automatiquement.

4.14.12. Permutation circulaire à droite d'un quartet

Syntaxe : `RegSRC.f`
Exemple : `ASRC.W`

4.14.13. Permutation circulaire à gauche d'un quartet

Syntaxe : `RegSLC.f`
Exemple : `ASLC.W`

4.14.14. ET logique

Syntaxe : `Reg1=Reg1&Reg2.f` ou `Reg1&Reg2.f`
Exemples : `C=C&B.A` ou `C&B.A`

4.14.15. OU logique

Syntaxe : `Reg1=Reg1!Reg2.f` ou `Reg1!Reg2.f`
Exemples : `C=C!B.A` ou `C!B.A`

4.14.16. NON logique

Syntaxe : `Reg=-Reg-1.f`
Exemple : `C=-C-1.A`

4.14.17. NON mathématique

Syntaxe : `Reg=-Reg.f`
Exemple : `C=-C.A`

4.14.18. Charger une valeur dans un registre de sauvegarde

Syntaxe : `SReg=Reg.f`
Exemple : `R0=A.W`
NB : `Reg` peut seulement être la registre `A` ou `B`

4.14.19. Charger une valeur dans A ou C à partir d'un registre de sauvegarde

Syntaxe : `Reg=Sreg.f`
Exemple : `A=R1.X`
NB : `Reg` peut seulement être la registre A ou B

4.14.20. Echanger A ou C avec un registre de sauvegarde

Syntaxe : `RegSRegEX.f`
Exemple : `AR1EX.X`
NB : `Reg` peut seulement être la registre A ou B

4.14.21. Ecriture en mémoire (POKE)

Ces instructions permettent d'écrire la valeur de A ou C à l'adresse où pointe `D0` ou `D1`.
Syntaxe : `Data=Reg.f` ou `Data=Reg.x`
Exemples : `DAT1=C.A` ou `DAT0=A.10`
NB : `Reg` peut seulement être la registre A ou B

4.14.22. Lecture en mémoire (PEEK)

Ces instructions chargent des données dans A ou C pointées par `D0` ou `D1`.
Syntaxe : `Reg=Data.f` ou `Reg=Data.x`
Exemples : `C=DAT1.A` ou `A=DAT0.10`
NB : `Reg` peut seulement être la registre A ou B

4.14.23. Applications avec D0 et D1

4.14.23.1. Chargement de D0 et D1

Syntaxe : `DReg=hh` ou `DReg=hhhh` ou `DReg=hhhhh`
ou `DReg=(2)Exp` ou `DReg=(4)Exp` ou `DReg=(5)Exp`
Exemples : `D0=FF  D1=12345 D1=(5)toto+$5`

4.14.23.2. Echanges entre A ou C et les pointeurs D0 ou D1

4.14.23.2.1. Charger A ou C champ A dans D0 ou D1

Syntaxe : `DReg=Reg`
Exemple : `D0=A`
NB : `Reg` peut seulement être la registre A ou B

4.14.23.2.2. Charger les 4 1ers quartets de A ou C dans D0 ou D1

Syntaxe : `DReg=RegS`
Exemple : `D0=AS`
NB : `Reg` peut seulement être la registre A ou B

4.14.23.2.3. Echange entre A ou C champ A et les pointeurs D0 ou D1

Syntaxe : `RegDRegEX`
Exemple : `AD1EX`
NB : `Reg` peut seulement être la registre A ou B

4.14.23.2.4. Echange entre les 4 1ers quartets de A ou c et les pointeurs D0 ou D1

Syntaxe : `RegDRegXS`
Exemple : `AD1XS`
NB : `Reg` peut seulement être la registre A ou B

4.14.23.3. Incrémenter et décrémenter D0 et D1

Syntaxe : `DReg=DReg+Cst` ou `DReg+Cst`
`DReg=DReg-Cst` ou `DReg-Cst`
Exemples : `D0=D0+12 D0-50`
NB1 : le Saturn n'est pas capable d'ajouter une constante plus grande que 16 à un pointeur, mais si `Cst` est plus grand alors MASD génèrera automatiquement les instructions nécessaires (exemple : `D0+20.A` donnera `D0+16.A D0+4.A`).
NB2 : même si on additionne souvent des constantes à des pointeurs, les grosses constantes doivent être évitées car elles ralentiront le programme et génèrerons aussi un gros programme.
NB3 : après avoir ajouté une constante plus grande que 16 à un pointeur, la retenue CARRY ne pourra être testée.
NB4 : vous pouvez aussi utiliser une expression plutôt qu'une constante (MASD doit être capable d'évaluer l'expression). Si l'expression est négative, MASD inversera l'addition en soustraction et vice versa.
NB5 : soyez prudent quand vous utilisez la soustraction, c'est facile de se tromper ! `D0-5-6` est équivalent à `D0+1` et non à `D0-11`.

4.14.24. Tests

NB1 : un test est toujours suivi par `RTNYES`, `GOYES`, `SKIPYES`, `EXIP`, `UP`, `GOTO`, `GOTOL`, `GOVLNG`, `GOSUB`, `GOSUBL` ou `GOSBVL`.
NB2 : `RTY` correspond à la même chose que `RTNYES`
NB3 : une flèche (¨) peut être suivie par un nom de label (elle remplace `GOYES`) ou peut être suivie par un saut de bloc, qui est l'équivalent à l'inverse du test suivi de `SKIPYES`, afin de reproduire une structure IF-THEN. Exemple : `?A=C.A -> { }` est le même que `?A#C.A { }`.
NB4 : `SKIPYES` peut être enlevé si le test est suivi par un saut de bloc (`{ }`).
NB5 : si le test est suivi par un `GOTO`, `GOTOL`, `GOVLNG`, `GOSUB`, `GOSUBL` ou `GOSBVL`, MASD compile l'inverse du test pour reproduire un `GOYES` avec une étendue plus grande. Exemple : `?A=C.A GOTO B` est la même chose que `?A#C.A { GOTO B }`.
NB6 : `GOTO`, `GOTOL`, `GOVLNG`, `GOSUB`, `GOSUBL`, `GOSBVL` ou ¨ { ne peuvent pas être utilisés après un test de type `HST`.
NB7 : un nom de label doit obligatoirement suivre `GOYES`, `GOTO`, `GOTOL`, `GOVLNG`, `GOSUB`, `GOSUBL` ou `GOSBVL`.

4.14.24.1. Tests d'égalité et de différence

Syntaxe : `?Reg1=Reg2.f ?Reg1#Reg2.f ?Reg1‹Reg2.f`
Exemples : `?A=C.B ?C#D.A ?C‹B.X`
NB : le caractère HP de l'inégalité peut être utilisé.

4.14.24.2. Tests de supériorité et d'infériorité

Syntaxe : `?Reg1<Reg2.f ?Reg1‰Reg2.f ?Reg1<=Reg2.f`
`?Reg1>Reg2.f ?Reg1ŠReg2.f ?Reg1>=Reg2.f`
Exemples : `?A<C.B ?C‰B.X ?C<=D.M`
`?A>C.B ?CŠB.X ?C>=D.M`

4.14.24.3. Tests de nullité

Syntaxe : `?Reg=0.f ?Reg#0.f ?Reg‹0.f`
Exemples : `?A=0.B ?B#0.X ?C‹0.A`

4.14.24.4. Tests sur certains bits des registres A et C

Syntaxe : `?RegBIT=v.a` où `Reg` est `A` ou `C`.
Exemple : `?ABIT=1.5 GOYES TOTO`

4.14.25. Opérations avec PC

```
A=PC  C=PC  PC=A  PC=C  APCEX  CPCEX  PC=(A)  PC=(C)
```

4.14.26. Opérations sur le registre STATUS

```
SB=0  XM=0  SR=0  MP=0  HST=a  ST=v.a
?SB=0  ?XM=0  ?SR=0  ?MP=0  ?HST=a  ?ST=v.a
```

4.14.27. Le champ P

```
P=a  P=P+1  P+1  P=P-1  P-1
?P=a  ?P<a  ?P#a
P=C.a  C=P.a  CPEX.a  C=C+P+1  C+P+1
```

4.14.28. Instructions de sauts

```
GOTO label
GOTOL label ou GOLONG label
GOVLNG Cst              où Cst est un nombre hexadécimal
GOVLNG =label           où label est une constante, ou un label en mode absolu
GOVLNG ="COMMAND"       COMMAND est une entrée de la table STARTEXT
GOSBVL Cst              où Cst est un nombre hexadécimal
GOSBVL =label           où label est une constante, ou un label en mode absolu
GOSBVL ="COMMAND"       COMMAND est une entrée de la table STARTEXT
GOSUB label
GOSUBL label
GOC label
GONC label
GOTOC label identique à SKIPNC { GOTO label }
GOTONC label identique à SKIPC { GOTO label }

RTN  RTNSXM  RTNCC  RTNSC  RTI  RTNC  RTNNC
RTNYES  RTY après un test.
```

4.14.29. Echanges entre C et RSTK

C=RSTK et RSTK=C sont des instructions qui permettent de poser et prendre des données dans la pile des retours du Saturn.

4.14.30. Instructions d'entrées/sorties

```
OUT=CS  OUT=C  A=IN  C=IN
```

NB1 : les instructions A=IN et C=IN sont bugguées (elles fonctionnent seulement sur certains adresses). Utilisez plutôt A=IN2 et C=IN2 qui font appel aux instructions A=IN et C=IN en ROM.
NB2 : OUT=C=IN est un appel en ROM de OUT=C C=IN

4.14.31. Contrôle du Saturn

Mode hexadécimal et décimal : SETHEX SETDEC
Autres instructions : UNCFNG CONFIG RESET C=ID SHUTDN INTON INTOFF RSI

4.14.32. Nouvelles instructions de MASD

GOINC label	Equivalent à LC(5)label-&. (& est l'adresse de l'instruction)
GOINA label	Equivalent à LA(5)label-&. (& est l'adresse de l'instruction)
$hhhh...hhh ou NIBHEX hhh..hh	Inclut des données hexadécimale. Exemple: $12ACD545680B.
$/hhhh...hhh	Inclut des données hexadécimale en ordre inverse. Exemple : $/123ABC est équivalent à $CBA321.
$(x)Exp ou CON(x)Exp ou	Place la valeur contenue dans Exp dans le Code sur x quartets.

EXP(x)Exp	
¢Ascii	Intègre des données ASCII. La fin de la chaîne est un autre ¢ ou un retour chariot. Exemple: ¢Hello¢. Pour saisir le caractère ¢, l'écrire deux fois. Pour écrire un caractère en fonction de sa valeur ASCII, utiliser \xx où xx est un nombre hexadécimal. Pour saisir un \, l'écrire deux fois aussi.
INCLUDE FileName	Inclut directement un fichier spécifié comme données.
GOIN5 lab G5 lab GOIN4 lab G4 lab GOIN3 lab G3 lab GOIN2 lab G2 lab	Comme $(x)label-& avec x=5, 4, 3 ou 2. Utile pour créer un saut de table.
SAVE	Equivalent à GOSBVL 0679B
LOAD	Equivalent à GOSBVL 067D2
RPL ou LOOP	Equivalent à A=DAT0.A D0+5 PC=(A)
LOADRPL	Equivalent à GOVLNG 05143
INTOFF2	Equivalent à GOSBVL 26791
INTON2	Equivalent à GOSBVL 26767
ERROR_C	Equivalent à GOSBVL 266C6
A=IN2	Equivalent à GOSBVL 0020A
C=IN2	Equivalent à GOSBVL 00212
OUT=C=IN	Equivalent à GOSBVL 0020F
RES.STR	Equivalent à GOSBVL 05B7D
RES.ROOM	Equivalent à GOSBVL 039BE
RES.RAM	Equivalent à GOSBVL 26920
SHRINK$	Equivalent à GOSBVL 039BE
COPY<- COPY¬ COPYUP	Equivalent à GOSBVL 0670C
COPY-> COPY¨ COPYDN	Equivalent à GOSBVL 066B9
DISP	Equivalent à GOSBVL 2685C (Seulement si debug est actif)
DISPKEY	Equivalent à GOSBVL 26863 (Seulement si debug est actif)
SHKLST	Equivalent à GOSBVL 039BE
SCREEN	Equivalent à GOSBVL 2677C
MENU	Equivalent à GOSBVL 26783
ZEROMEM	Equivalent à GOSBVL 0675C
MULT.A	Equivalent à GOSBVL 03991
MULT	Equivalent à GOSBVL 2709E
DIV.A	Equivalent à GOSBVL 03F24
DIV	Equivalent à GOSBVL 26EA4
BEEP	Equivalent à GOSBVL 267F3

4.15. Commandes MASD

!PATH+ *DirName*	Ajoute les répertoires donnés dans la liste de recherche des chemins.
!OFF	Eteint l'écran de la machine, ce qui permet un gain de vitesse de 13% lors de l'assemblage.
!WAROFF	MASD n'affichera pas les labels fantômes
!NO CODE	MASD ne génèrera pas le prologue $02DCC mais donnera directement les données.
!DBGON	MASD génèrera un code quand DISP ou DISPKEY seront trouvés dans la source.
!DBGOFF	MASD ne génèrera pas de code quand DISP ou DISPKEY seront trouvés dans la source.
!1-16	Bascule en mode 1-16
!0-15	Bascule en mode 0-15
!RPL	Bascule en mode RPL.
!ASM	Bascule en mode ASM.

!FL=0.*a*	Efface le flag de compilation a.
!FL=1.*a*	Arme le flag de compilation a.
!?FL=0.*a*	Compile la fin de la ligne si le flag *a* est armé.
!?FL=1.*a*	Compile la fin de la ligne si le flag *a* est désarmé.
!ABSOLUT *Addr*	Bascule en mode absolu. Le programme commence à *Addr*. Note: MASD considère toujours que le prologue $02DCC et la longueur du code sont au début du programme même si !NO CODE figure dans la source.
!ABSADR *Addr*	En mode absolu, ajoute des quartets vides pour aller à l'adresse demandée. Si c'est impossible, on obtient une erreur.
!EVEN	En mode absolu, une erreur peut être provoquée si la commande n'est pas sur une même adresse.
!ADR	MASD génèrera une source en définissant des constantes et labels utilisés dans le programme à la place du programme.
!COMPEXP	Oblige MASD à calculer toutes les expressions précédentes.

4.16. Les messages d'erreur

Invalid File	Le fichier n'est ni une source ni une macro (il doit finir @).
Too many links	Seuls 256 links sont supportés. Peut-être dû à une récursivité.
Unknown Instr.	Instruction inconnue.
Bad Field	Champ inconnu.
0-15 Expected	Un entier entre 0 et 15 est attendu.
1-16 Expected	Un entier entre 1 et 16 est attendu.
Error Stack Size	Vous utilisez plus de 3 parenthèses.
Size Too Small	Le programme entre parenthèses est trop grand.
Close Size	Il y a plus de] que de [.
Label Expected	Un nom de label est attendu.
Const Expected	Un nombre hexadécimal est attendu.
Can't Find	Ne trouve pas le label ou le fichier.
Lab. Already Dec.	Un label ou une constante est déjà déclaré avec le même nom.
Need {	Ce crochet { est attendu.
Need }	Ce crochet } est attendu.
Forbidden	Opération interdite.
Bad Expression	Erreur dans l'expression.
Jump too Long	Saut trop grand.

4.17. Exemple : animation laser

Le programme qui suit prend un objet graphique de dimensions 131x64 sur le niveau de la pile et effectue une gravure laser de l'image. Voici la source :

```
"!RPL !NO CODE
::
TURNMENUOFF

CODE

 SAVE ST=0.15 %pas d'interruptions
 A=DAT1.A D0=A LC 00014 A+C.A R4=A.A % adr 1ers pixels du grob
 A=DAT0.A LC 02B1E
 ?A#C.A ¨ FIN % teste le prologue
 D0+10 A=0.W A=DAT0.10 C=0.W LC 8300040
 ?A=C.W ¨ OK % teste la taille

*FIN ST=1.15 GOSBVL 267A6 LOADRPL

%En 80092 se trouvent les coordonnées courantes du point à tester.
%Dans le registre R3.A se trouve le début de l'écran courant.
%Le flag 2 correspond à la couleur du laser (0=noir, 1=blanc).
%Le flag 3 correspond au sens de balayage du laser (0=gauche, 1=droite).
```

```
*OK
 D0=80092 LC 3F82 DAT0=C.4
 GOSBVL 2677C R3=A.A % sauve l'écran
 D1=A LC 00880 GOSBVL 0675C % efface l'écran
 ST=0.2 ST=0.3 D0=00F9 LC 0041 DAT0=C.4

%A chaque tour de boucle, on regarde si la touche est DROP (out=010,
in=0001).

*DEBUT
 LC 001 OUT=C=IN
 ?CBIT=1.6 ¨ FIN %touche DROP
 D0=80094
 A=0.A A=DAT0.B D0-2 C=0.A C=DAT0.B A+A.X B=A.A ASL.A B+A.A
 CSRB.B CSRB.B B+C.A C+C.B C+C.B A=R4.A A+B.A D1=A A=DAT0.B
 A-C.B C=DAT1.1

*TEST.PT
 ?A=0.B ¨ TEST.PIX
 A-1.B CSRB.B GOTO TEST.PT

*TEST.PIX
 ?CBIT=0.0 ¨ INCREM
 GOSUB SAUVE.L GOSUB LIGNE ST=1.2 GOSUB LIGNE ST=0.2 GOSUB RECUP.L
 D0=80092
 A=DAT0.4 D0=0101 DAT0=A.4 ST=0.0 D1=80101 GOSUB POINT

*INCREM
 D0=80092 A=0.A A=DAT0.4 C=0.A LC 82
 ?A#C.A ¨ SU.INC
 GOTO FIN

*SU.INC
 ?ST=1.3 ¨ I2
 ?A=0 B ¨ MONTED
 A-1.B DAT0=A.B GOTO DEBUT

*MONTED GOSUB MONTE ST=1.3 GOTO DEBUT

*I2 ?A=C.B ¨ MONTEG
 A+1.B DAT0=A.B GOTO DEBUT

*MONTEG GOSUB MONTE ST=0.3 GOTO DEBUT

*MONTE D0+2 A=DAT0.B A-1.B DAT0=A.B RTN

*SAUVE.L GOSUB LIGS D0=80200 D1=A GOSUB COPY RTN

*RECUP.L GOSUB LIGS D1=80200 D0=A GOSUB COPY RTN

*LIGS D0=80094 A=0.A A=DAT0.B A+A.A C=A.A CSL.A
 C+A.A A=R3.A A+C.A RTN

*COPY LC 10 { A=DAT1.B DAT0=A.B D1+2 D0+2 C-1.B UPNC } RTN

*LIGNE
 D0=80092 A=DAT0.4 D0=F5 DAT0=A.4 ST=0.0 ST=0.1 D1=800F5
 GOSUB DXDY D1-6 GOSUB DXDY C=DAT1.B D1-2 A=DAT1.B
 ?A>C.B ¨ NOINV2
 ACEX.B ST=1.0
*NOINV2
 DAT1=A.B D1+2 DAT1=C.B D1-6 C=DAT1.A D1-4 A=DAT1.A D1+16
 DAT1=C.4 D1-4 DAT1=A.4
 ?ST=0.0 ¨ NOINV3
 C=A.A ASR.A ASR.X DAT1=A.B D1+2 DAT1=C.B D1+2 A=DAT1.4
 C=A.A ASR.A ASR.X DAT1=A.B D1+2 DAT1=C.B D1-6 A=DAT1.A
 D1+4 C=DAT1.A GOTO SU1
*NOINV3 D1+4
*SU1 ?A<C.B ¨ NOINV4
 ACEX.A
*NOINV4 DAT1=C.4 D1-4 DAT1=A.4 ASR.A ASR.X CSR.A CSR.X
 ?A<C.B ¨ NOINV5
```

```
 ST=1.1
*NOINV5
 D1-2 C=0.X C=DAT1.B C+C.X D1-2 A=0.X A=DAT1.B C-A.X
 D1+10 DAT1=C.X D1-6 GOSUB POINT
*BOUCLE
 C=DAT1.B D1-4 A=DAT1.B
 ?A=C B RTY
 A+1.B DAT1=A.B D1+6 A=DAT1.X D1-8 C=0.X C=DAT1.B C+C.X
 ?ABIT=0.11 ¨ SUITE
 A+C.X D1+8 GOTO SUITE3
*SUITE
 A+C.X C=0.X D1-2 C=DAT1.B C+C.X A-C.X
 D1+6 C=DAT1.B
 ?ST=0.1 ¨ INCR
 C-1.B GOTO SUITE2
*INCR C+1.B
*SUITE2 DAT1=C.B D1+4
*SUITE3 DAT1=A.X D1-6 GOSUB POINT GOTO BOUCLE

*DXDY A=DAT1.B D1+4 C=DAT1.B A-C.B GONC NOINV A=-A.B
*NOINV D1+4 DAT1=A.B RTN

*POINT
 A=0.A C=0.A A=DAT1.B D1+2 C=DAT1.B
 ?ST=0.0 ¨ NOINV6
 ACEX.A
*NOINV6
 D1+12 DAT1=A.A D1+5 DAT1=C.A C+C.A A=C.A CSL.A C+A.A A=R3.A
 A+C.A D1-5 C=DAT1.A CSRB.A CSRB.A A+C.A D0=A C+C.A C+C.A
 A=DAT1.A C=A-C.A LA 01
 ?C=0.B ¨ FINPT
 C-1.B { A+A.B C-1.B UPNC }
*FINPT
 C=DAT0.1
 ?ST=0.2 ¨ NRM
 A=-A-1.B A&C.B C=0.B
*NRM
 A!C.B DAT0=A.1 D1-10 RTN

ENDCODE
;

@"
```

Et revoici le même programme revu, corrigé et (largement) raccourci par Sébastien Casartelli :

```
!NO CODE !RPL
CK1&Dispatch grob
::
TURNMENUOFF
CODE

% R0a: X
% R1a: Y
% R2a: adresse grob

SAVE GOSBVL DisableIntr
A=DAT1.A D0=A LC 00014 A+C.A R2=A.A
D0+10 A=0.W A=DAT0.10 C=0.W LC 8300040 ?A=C.W
{ *.End GOSBVL AllowIntr LOADRPL }

GOSBVL "D0->Row1" D1=A D0-15 C=DAT0.A C-15.A GOSBVL WIPEOUT
LC 0003F R1=C.W

{
  LC 00082
  {
    R0=C.A
    LC 001 GOSBVL OUTCINRTN ?CBIT=1.6 -> .End
    GOSUB .PointAndLine
    C=R0.A C-1.A UPNC
```

```
  }
  A=R1.W A-1.A R1=A.A
  {
    LC 001 GOSBVL OUTCINRTN ?CBIT=1.6 -> .End
    GOSUB .PointAndLine
    A=R0.A A+1.A R0=A.A LC 83 ?A#C.B UP
  }
  A=R1.A A-1.A R1=A.A UPNC
}
GOTO .End

*.PointAndLine
A=R1.A A+A.A C=R2.A C+A.A ASL.A A+C.A
C=R0.A P=C.0 CSRB.A CSRB.A A+C.A D0=A
LC 2481248124812481 P=0
A=DAT0.B A&C.P ?A=0.P RTY
GOSUB LIGNE GOSUB LIGNE
GOSBVL "D0->Row1" D0-20
A=R0.A C=R1.A GOVLNG aPixonB

*LIGNE
GOSBVL "D0->Row1" D0-20
A=R0.A B=A.A LA 00041
C=R1.A D=C.A C=0.A
GOVLNG aLineXor

ENDCODE
;
;
@
```

5. Le Désassembleur

Le désassembleur convertit des codes binaires en sources sous forme de chaînes hexadécimales, grâce à la commande ASM¨. Il y a deux possibilités pour l'utiliser : avec un Code sur le niveau 1 de la pile ou avec deux adresses (sous forme d'entiers binaires) dont l'une pointe le début du code à désassembler et l'autre la fin.
La syntaxe utilise celle de MASD en mode 0-15.
Chaque ligne contient une adresse et une instruction.
Si le flag –71 est armé (en utilisant -71 SF) les adresses ne sont pas affichées, sauf les labels et les destinations de sauts. Dans ce cas, le résultat de la source peut être réassemblé si nécessaire.

Exemple:

-71 CF (defaut)	**-71 SF**
AE734 GOSBVL 0679B AE73B LC 01000 AE742 C-A A AE744 GONC AE742 AE747 GOVLNG 138B9	GOSBVL 0679B LC 01000 *AE742 C-A A GONC AE742 GOVLNG 138B9

Le programme ASM¨ prend :
- soit un objet Code sur le niveau 1 de la pile,
- soient deux entiers binaires sur les niveaux 1 et 2 de la pile.

6. La library des points d'entrée

La library des points d'entrée est une library externe (que vous pouvez télécharger sur le site web de Hewlett Packard) qui contient la table des noms et adresses des points d'entrée. C'est utilisé par l'assembleur MASD pour relever les valeurs des points d'entrée assembleur et Rpl-système (comme par exemple TURNMENUOFF en Rpl-système).

Cette library peut être stockée en port 0, 1 ou 2.
Et si vous désirez programmer en Rpl-système, vous devez installer cette library.

Cette library contient 4 fonctions :

6.1. nop

Cette fonction est ici utilisée en interne et ne devrait pas être utilisée.
Exécuter cette fonction ne fait rien.

6.2. GETNAME

Donne le nom de l'adresse d'un point d'entrée.
Exemple : `# 054AFh GETNAME` donne `"INNERCOMP"`.
NB : comme plusieurs entrées peuvent avoir la même adresse, GETNAME n'est pas une fonction bijective.

1: entier binaire ⇒ 1: chaîne de caractères

6.3. GETADR

Donne l'adresse du nom d'un point d'entrée.
Exemple : `"INNERCOMP" GETADR` donne `# 054AFh`.
NB : comme plusieurs entrées peuvent avoir la même adresse, GETNAME n'est pas une fonction bijective.

1: chaîne de caractères ⇒ 1: entier binaire

6.4. GETNAMES

Cherche toutes les entrées dont le nom commence par une chaîne de caractères spécifique.
NB : donner une chaîne de caractères vide en entrée retournera la liste de tous les points d'entrée !

1: chaîne de caractères ⇒ 1: liste des noms des entrées

7. Contacts

Pour toute modification du logiciel (donc de la ROM de la machine), veuillez trouver le détail des nouveautés sur le site officiel HP : **`www.hp.com/calculators/france`**.
Retrouvez la documentation de MASD sur : **`home.nordnet.fr/~phpamart`** **rubrique "Infos 49".**